Straight to the Heart of Computer Science via Functional Programming

Robert Giegerich[*] *Ralf Hinze*[†] *Stefan Kurtz*[‡]

Abstract

We outline a deductive concept for an introductory course to computer science aimed at CS students as well as students from other disciplines. The emphasis is on introducing fundamental concepts of computer science and showing how they evolve from each other. Functional programming in *Haskell* is used as a vehicle to explain these concepts and to allow for practical exercises. We argue that functional programming is more than just a good option for such an introductory course. Functional programming is so intimately related to the essence of computing that it allows to develop CS fundamentals in a way that is logically stringent as well as exciting.

1 Rationale of the Bielefeld Introduction to CS

1.1 FP as a Vehicle for Studying CS Fundamentals

All computer science curricula have to make a fundamental decision about how beginners should be introduced to CS. Some start with a programming course aiming to give emphasis to practical experience before theoretical underpinning. Some start with theoretical aspects of computing postponing programming to later or concurrent courses. While both ways are feasible (after all, the authors were taught CS that way), both have a disadvantage: The first suggests to the student that one can become a good programmer without studying theory. The second separates formal concepts from programming practice, suggesting the impression that programming theory is something else, but not the explanation of what we do in practice. The student's attitude to theory as "mere theory" results from this, and both, students and teachers, later pay a high price for this. The best solution, of course, is to do both kinds of courses in parallel. But then, the same instructor should teach both courses, otherwise the problem is only disguised.

The Bielefeld computer science department runs a non-standard CS program called "Naturwissenschaftliche Informatik", where students take two major subjects of about equal proportions, say, CS and molecular biology. The term "Bioinformatics" has been coined for this particular new branch of science. Together with governmentally imposed restrictions on the volume of study, the combination of two major subjects requires considerable efforts to design a manageable curricular volume. At the same time, the introductory course is always attended by students from a wide spectrum of other fields. This may be the only course in computer science they take. So we really want to convey a fundamental understanding.

1.2 Overview of Topics

Although our CS-course does not explicitly dwell on this, it is based on a very strict line of deductive reasoning. We begin with a definition of computer science and then we see were this leads us to. An overview of the topics is given in Figure 1.

Since the pathways of logical deduction and those of learning do not always coincide, topics are not taught strictly in the order shown in Figure 1. The course postpones the topics 2c, 4d, and 4e when moving through the corresponding chapters. There are two good reasons for doing so: Languages and grammars are better motivated after some programming has been done and arrays (as well as dynamic programming) only arise because of efficiency concerns. After the chapter on efficiency, there is a "virtual" chapter where we return to modelling, programming methods, and efficiency concerns in an amalgamated fashion. We introduce grammars, parser combinators, and their transformation into dy-

[*]Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, Email: robert@techfak.uni-bielefeld.de

[†]Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany, Email: ralf@informatik.uni-bonn.de

[‡]Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, Email: kurtz@techfak.uni-bielefeld.de, partially supported by DFG-grant Ku 1257/1-1

1. Theory of Science
   (a) What is Computer Science?
   (b) Traditional fields of CS
   (c) Relation to other sciences
2. Modelling
   (a) Music
   (b) Molecular Genetics
   (c) Grammars, Languages, and Recognizers
3. A Programming Language
   (a) Types
   (b) Equations
   (c) Expressions
   (d) Computing in *Haskell*
   (e) Application: Binary Trees
4. Programming Methods
   (a) Structural Recursion
   (b) Divide-and-Conquer
   (c) Application: Sorting
   (d) Combinator Languages
   (e) Dynamic Programming
5. Algorithmic Complexity
   (a) Asymptotic Efficiency
   (b) Problem Complexity
   (c) Program Optimization
6. Abstraction
   (a) Type Classes
   (b) Sequence Data Types
   (c) Outlook: Object Oriented Programming

Figure 1: The topics of our CS-course

namic programming algorithms.

1.3 Goals of this Paper

In the remainder of this short paper, we cannot redraw the complete concept of our course. We shall be very short on those subjects that are taught fairly conventionally and a little more explicit on those that have a special touch. We give some examples how the subjects we are teaching are conveniently expressed in *Haskell* [4]. Some details are omitted and we rely on the reader to be able to understand the *Haskell* examples without much added explanation. We hope that these examples provide convincing evidence that functional programming is an excellent vehicle to take the students' minds straight to the heart of computer science.

2 Fundamental Concepts of Computing

The following is a condensed version of the main topics of our CS-course.

2.1 The Science of Computation by Machine

Computer Science is the science of computation by machine. What is special about computation such that it can be done by a machine? We observe ourselves when we do numerical calculation: We restrict our thinking to formal reasoning, more precisely, we manipulate numbers or more generally formulae according to rules of arithmetic or algebra. Why these rules are adequate does not concern us during calculation. In fact, restricting our attention to obeying these rules makes calculation both effective and boring. By its very nature, computing with numbers is a mechanical exercise of the mind. To transfer this to a machine, all it requires is a device that can manipulate formulae in a reliable and flexible manner. No wonder that the first numeric calculators were invented shortly after mankind had learned to divide and long before engineering was advanced enough to build such devices reliably.

Once such devices have been built, a fundamentally new question enters the horizon of science — programming. Now there is a machine that can do formal manipulation, what can we have it do? Aside from numeric computation, which problems of reality can be modelled in terms of formulae (data structures) and rules (programs)? There are few areas of interest that are, like arithmetic, formal by nature; in all other cases our formalization only approximates reality. This accounts for the difficulty as well as for the excitement of programming.

Theoretical Computer Science studies the absolute (decidability) and gradual limits (complexity) of what can be computed by machine. Its most important finding is that all machines with a minimal set of capabilities can solve the same class of problems. (That there are formally undecidable problems does not come as a surprise to anyone who started out understanding computing as a restricted sort of reasoning.) The consequence of Church's thesis on the technical side is that *Technical Computer Science* concentrates on making computers faster, smaller, and cheaper while there is no evolution in the basic operations. With computers remaining as primitive as ever, the burden is on *Practical Computer Science*: It has to bridge the semantic gap between the machine's capabilities and the ever more advanced applications by building layers of abstraction, such as operating systems, programming languages, network protocols, and the World Wide Web. *Applied Computer Science* uses these achievements to extend computer usage into all sciences and all aspects of human life. The use of computers to do tasks also done by humans tends to create the illusion that the job is done in the same way. The illusion of *Artificial Intelligence* has been the halo of computers ever since the sixties. As a research field dedicated to the promotion of the illusion that comes from a subject (rather than the subject itself), the advance of Artificial Intelligence adheres to a pattern of promise and disappointment: while its best results are absorbed by the general advance of computer science, the illusion is never perfect.

As much as everyone, computer scientists should be aware of the social impacts of their work. Although it makes us feel important, computer scientists should reject the modernistic view of living in "the computer age". The rapid pervasion of the working sector has a social as much as a technical reason. By the advent of the computer, the disintegration of labor had already created a wide area of jobs stripped off completely of creativity and responsibility, with workers largely reduced to mindful machines. It takes no "computer revolution" to replace them by software and to concentrate the more demanding tasks on a few remaining jobs. That the computer is everywhere does not mean it is the driving force of our society. Computer scientists taking social responsibility must be prepared to study economy and politics as well as computer science.

2.2 The Challenge of Modelling

If the central task of CS is approximating aspects of reality by formal models, then we should emphasize the importance of modelling. Modelling goes before computation in our course. This reverses current approaches, which start with computation on trivial (not to say boring) data and only add data structures and real world examples at a rather late stage. Our dogma of Chapter 1 (Theory of Science) is "data are formulae" and, indeed, we can look at modelling right away by constructing worlds of formulae. The difficulty is to find real world subjects that are somewhat interesting but still easy to model. The solution is to choose subjects that by their own nature have a largely formal character. They should be easy to cast into formulae but still rich enough to be interesting. (This is why numbers do not qualify here as they are a completely formal world already and modelling is trivial.) Fortunately, there are such subjects. *Music* and *molecular genetics* are our two introductory subjects.

In spite of the artistic expression, individual performance, and other aspects, music has a largely formal character. The harmonic scale, polyphony, and rhythm follow formal rules. So does the structure of musical pieces, be they a simple folk tune or a symphony. Consequently, a formal notation has already evolved to describe music. While traditional musical scores are written in a graphical notation, we introduce formulae to describe notes, chords, durations, harmonic inversion etc. leading to a formula that describes a complete piece of music, see Figure 2. The music formulae are written in *Haskell* notation borrowed from the *Haskore* system [2].

As a second semi-formal system found in nature we use molecular genetics. The cell's "data" are represented by long chain molecules built from nucleotides or amino acids—nature uses strings over two different alphabets. We model enzyme activities as functions (DNA replication, proof reading). The ribosome constitutes a function that applies the genetic code (another function) to split messenger RNA into triplets and to produce the corresponding amino acid sequence. We also discuss the limits of our model. For example, the cell can distinguish a fresh complementary copy of some DNA strand from its original—in our model, once calculated, they are merely two strings without history, see Figure 3.

Music and molecular genetics are certainly not the standard examples for introduction to CS. Most textbooks prefer more abstract data types like numbers or lists. It is the semiformal nature of the chosen topics which makes these examples workable requiring no more than average high school knowledge about music and genetics. Choosing such examples also places students with diverse backgrounds in computing on the same level.

At the end of the modelling chapter we have a number of interesting data types to play with as well as first examples of type polymorphism and higher order functions. We have not talked about computing yet and some students are actually taken by surprise when *Hugs* comes in a week later to execute the `ribosome` function.

2.3 A Simple Language for Programming

The programming language *Haskell* is introduced in a rather rudimentary form. We fix the syntax for data type declarations, expressions, and equations. We skip many notational conveniences of *Haskell*. We try to restrict the specific teaching of the language to about five sessions. This works because students already know most of the notation from the modelling chapter. Evaluation order is left arbitrary. We assure the students that the computer calculates with these formulae and equations the same way that they do, except more reliably by following formally stated rules of reduction. Funny enough, the critical point in this chapter is to convince the students that there is no further magic behind computing by machine than formula manipulation guided by equational definitions.

2.4 Programming Methods and Reasoning about Programs

As general methods of programming we introduce *Structural Recursion* and *Divide-and-Conquer*. Program examples from the previous chapter are analyzed and the recursion scheme is made explicit, see Figure 4. It is quite convenient that recursion schemes can be expressed in *Haskell* as higher order functions. This proves that the schemes are not just in the eye of the beholder but are working techniques even if we do not always make them explicit in our programming.

All along the way we have been doing proofs of simple program properties by equational reasoning. Structural and well-founded induction are formulated in this chapter as the proof rules corresponding to the two recursion schemes.

2.5 Asymptotic Efficiency

In everyday life (including that of the scientist) the term "complexity" is mostly used in a careless fashion. Our own lack of understanding of a subject is attributed to the subject as one of *its* properties. But we all—and hopefully especially our students—make the experience that what seems complicated at first glance turns into a convenience of reasoning after some time of study. In most cases, "complexity" is an excuse rather than a statement about anything. Thus it needs to be explained why in computer science algorithmic complexity is a meaningful notion. It does not mean that a program is hard to understand but it means that the machine that eventually has to execute it cannot do faster than a certain number of steps. Computational complexity is meaningful only because of Church's thesis, which says that computing machines do not become enlightened.

We introduce the typical $O(f(n))$ machinery. Time is measured by the number of reductions while space is measured by the size of the formula. The latter is visualized by the management of blackboard area of the lecturer performing a computation. Problem complexity is introduced via the example of decision trees for sorting including the assumptions made on basic operations that are possible on the data (sorting by comparison versus counting sort).

2.6 Modelling, Programming, and Efficiency Revisited

Reality modelling, programming, and efficiency are addressed once more in a more demanding setting. We study pairwise sequence comparison motivated by the study of phylogenetic relationships on (say) related genes from different species. (In the somewhat more abstract terms of computer science, this is the edit distance problem on strings.) We introduce a data type for sequence alignments and show that it is not expressive enough to model our intents. It allows to express various alignments that do not make sense biologically. A language of well-formed alignments is designated by a tree grammar. Determining all and only the well-formed alignments of two sequences is a parsing problem. Alignments are evaluated in terms of a scoring scheme for replacements, deletions, and insertions. Selecting the alignment(s) with a minimal edit distance is our first case of an optimization problem. So much for the problem specification.

Using *Haskell* infix operators a notation for tree grammars is introduced. A grammar turns into a recognizer by defining these operators as parser combinators [3]. This yields a recursive, predictive parser and an impressive efficiency problem as the number of alignments is exponential in the length of the sequences and some sub-alignments are re-calculated an exponential number of times. We then introduce arrays and augment the grammar with respect to tabulation of intermediate parser results. At the same time, we replace enumeration of alignments by their evaluation in terms of the distance score. This yields a program with polynomial time efficiency, see Figure 5. By means of partial evaluation, we derive the recurrences in terms of which dynamic programming algorithms are traditionally described in the literature, see [5].

Aside from the modelling chapter, this is the most unconventional topic in our course. An application of this approach to dynamic programming in a research context can be found in a companion paper [1] presented at the WAAAPL workshop at this conference.

2.7 Abstraction

The final chapter discusses various kinds of abstractions and how they help us to master the complexity of the formal models of reality that we create. It ends with an outlook on type classes and object oriented programming preparing the ground for teaching Java in the second semester course.

3 Conclusion

The approach outlined here allows us to cover a quite conventional selection of CS fundamentals in a somewhat unconventional way and in a very short time. The line of thought is equally challenging for novices as for students with considerable (imperative) programming experience. While we are quite happy with the course in its current form, it does create a problem for the second semester course: Students get bored because trivial programming problems, much simpler than those in the first course, take so much more writing and effort in an imperative setting.

References

[1] R. Giegerich, S. Kurtz, and G.F. Weiller. An Algebraic Dynamic Programming Approach to the Analysis of Recombinant DNA Sequences. In *Workshop on Algorithmic Ascpects of Advanced Programming Languages (WAAAPL)*, 1999.

[2] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore Music Notation. *J. Functional Programming*, **6**(3), 1996.

[3] G. Hutton. Higher Order Functions for Parsing. *J. Functional Programming*, **3**(2):323–343, 1992.

[4] S. Peyton Jones and J. Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, 1999. Available from `http://www.haskell.org/onlinereport/`.

[5] M.S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman Hall, 1995.

```
type Tone       =  Int
type Duration   =  Rational
data Instrument =  Oboe | HonkyTonkPiano | Cello | VoiceAahs
data Music      =  Note Tone Duration
                |  Pause Duration
                |  Music :*: Music            -- sequentiell composition
                |  Music :+: Music            -- parallel composition
                |  Instr Instrument Music
                |  Tempo Int Music

cDurScale =  Tempo allegro (c′ (3/8) :*: d′ (1/8) :*: e′ (3/8) :*: f′  (1/8) :*: g′ (1/8)
                                     :*: a′ (1/4) :*: h′ (1/8) :*: c′′ (1/2))

phrase1 =  c′ (1/4) :*: d′ (1/4) :*: e′ (1/4) :*: c′ (1/4)
phrase2 =  e′ (1/4) :*: f′ (1/4) :*: g′ (1/2)
phrase3 =  g′ (1/8) :*: a′ (1/8) :*: g′ (1/8) :*: f′ (1/8) :*: e′ (1/4) :*: c′ (1/4)
phrase4 =  c′ (1/4) :*: (transpose (-12) (g′ (1/4))) :*: c′′ (1/2)

verse        =  rep phrase1 :*: rep phrase2 :*: rep phrase3 :*: rep phrase4
infinite     =  verse :*: infinite
brotherJacob =  Tempo andante (Instr VoiceAahs
                    (entry (0/1) infinite :+: entry (2/1) (transpose 12 infinite) :+:
                     entry (4/1) infinite :+: entry (6/1) infinite))
```

Figure 2: Data type for music; a rhythmically swinging C-major scale; the canon “Brother Jacob” for four voices

```
data Nucleotide =  A | C | G | T

data AminoAcid  =  Asn     -- Asparagin
                |  Lys     -- Lysin
                |  ...     -- and so forth

type DNA     =  [Nucleotide]
type Protein =  [AminoAcid]
type Codon   =  (Nucleotide, Nucleotide, Nucleotide)

genCode           :: Codon -> AminoAcid
genCode (A, A, A) =  Lys;  genCode (A, A, G) =  Lys;  genCode (A, A, C) =  Asn  -- and so forth

ribosome                 :: DNA -> Protein -- the ribosome always starts at ATG
ribosome (A : T : G : x) =  Met : map genCode (triplets x)

triplets []              =  []
triplets (a : b : c : x) =  (a, b, c) : triplets x

wc_compl A =  T;  wc_compl T =  A;  wc_compl C =  G;  wc_compl G =  C

complSingleStrand []      =  []
complSingleStrand (a : x) =  wc_compl a : complSingleStrand x

dnaPolymerase x =  (x, complSingleStrand x)
```

Figure 3: The data types of the living cell; the genetic code and its use in gene translation; Watson-Crick complement and DNA polymerisation

```
list_recursion :: a -> (b -> [b] -> a -> a) -> [b] -> a
list_recursion base extend  = rec
    where rec []      =  base
          rec (a : x) =  extend a x (rec x)

insert   :: (Ord a) => a -> [a] -> [a]
insert a =  list_recursion [a] step
    where step b x sol =  if a <= b then a : b : x else b : sol
```

Figure 4: The scheme of structural recursion on lists

```
dp_editdistance alg x y =  axiom (alignment!)
    where (nil, repscore, delscore, insscore, select) =  alg

          alignment =  tabulated (
                           empty nil                                      |||
                           repscore  <<< xbase        -~~ (alignment!) ~~- ybase |||
                           delscore  <<< xbase        -~~ (alignment!)           |||
                           insscore  <<< (alignment!) ~~- ybase ... select)
```

Figure 5: The dynamic programming algorithm for computing the edit distance of two sequences; `select` is a choice function provided by the scoring algebra.